

A Generic Approach to Performance Modeling and Its Simulator Generator

*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*

by

V. Rajesh



to the

Department of Computer Science & Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July, 1998

05 JAN 1999 / ESE

CENTRAL LIBRARY
I. I. T., KANPUR

Vol. No. A

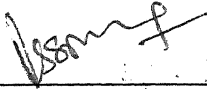
126865



A126865

Certificate

Certified that the work contained in the thesis entitled
“ *A Generic Approach to Performance Modeling and Its Application to Simulator Generator*”, by Mr.V. Rajesh, has been
carried out under my supervision and that this work has not
been submitted elsewhere for a degree.



Dr. Rajat Moona
Associate Professor,
Dept. of Computer Science and Engineering,
IIT Kanpur.

July, 1998

Abstract

An integrated environment for performance modeling is desirable in the present era of specification driven system design. With the complexity and the number of processors increasing rapidly, it is important to have a generic environment wherein the performance can be measured for a given processor on a given application. In this thesis, a simple, elegant and powerful language Sim-nML has been designed for performance modeling. Sim-nML acts as the specification language for a processor performance model in a generic way. As part of this thesis work, a generator for processor instruction set simulator is designed and implemented. The simulator generator takes Sim-nML specifications of a uniprocessor machine as input and produces code for the simulator. The performance simulator thus generated is an important tool in system design environment as it is useful to study the performance impact due to various design trade-offs.

Acknowledgments

The greatest debt of gratitude, I reserve for Dr. Rajat Moona who guided me at every stage of this project with his perspicacious suggestions, whose qualities have attracted me a lot. I thank the almighty for giving such a brotherly figure as my guide.

I express my gratitude to Cadence India Ltd. for their enduring support to this work, without which it must have been very difficult to complete this work.

I express my heart-felt thanks to all the faculty members for teaching the principles in most exciting and enjoyable way. I wish to thank my old teachers Dr. B. D. Chaudry, Dr. Krishna Kant and Miss Shantha Alis for being the source of inspiration for me. I wish to express my gratitude to Mr. Sakthi Shankar and Mr. Brahmaji Rao, for helping me out in crucial situations.

I am greatly indebted to my MTech friends Kataru, Dhanabal, Ramaswamy, Raja, Balaji, Gomes, Murali, Joye, Senthana, Venkat and all my BTech friends especially Aravindan, Nori, Ravi, Anand Kammanavar, Madan, Visu, Ezhil and Thambies for being affectionate and encouraging. Without them my college life must have got deserted. I thank all my MTech96 batch-mates especially Yugandhar, Kshitiz, Atul for their exciting company. Finally, I wish I could express my thankfulness in words, to my parents, my family, Pavithra and Sahana for their love and affection I have been receiving.

Contents

Acknowledgments	i
1 Introduction	1
1.1 Motivation	2
1.2 Overview of Related Works	3
1.3 Goals Achieved	3
1.4 Organization of Report	4
2 Performance Analysis	5
2.1 Salient Features of Modern Microprocessors	5
2.2 Simulation Methods	7
2.2.1 Depending on Resource Management	7
2.2.2 Depending on Instruction Sequence	8
2.3 A survey of Performance Analysis Tools	9
2.3.1 Alpha Microprocessor Model	9
2.3.2 SimOS	9
2.3.3 VMW	9
2.3.4 nML	10
3 Sim-nML Syntax and Semantics	13

3.1	Resource Usage Model	13
3.2	Specification of resource usage model using uses attribute	15
3.2.1	Uses Grammar	15
3.2.2	Implicit Parallelism	19
3.2.3	Specification of Feed Forwarding	22
3.2.4	Specification of Memory Hierarchy	22
3.2.5	Exception Declaration	23
3.2.6	Instruction Identification	24
3.2.7	Initial State and reg Declaration	24
3.3	Specification of Superscalar Processor and Branch Prediction	25
4	Implementation	30
4.1	Design of Simulator	30
4.1.1	Capturing Machine State	31
4.1.2	Capturing Hierarchical Structure	34
4.2	Implementation of Simulator Generator	39
4.2.1	Phase 1 - Gathering Hierarchical Information	40
4.2.2	Phase 2 - Type Detection	40
4.2.3	Phase 3 - Code Generation	41
4.3	Implementation of nML Type System	42
4.4	Generic Simulator Library	43
5	Results and Conclusion	44
5.1	Performance Models with Sim-nML	44
5.1.1	Generating Performance Simulator	44
5.1.2	Results	45
5.2	Conclusion	46

Chapter 1

Introduction

In design applications for embedded controllers, specification driven design automation is gaining momentum. It provides a fast turn-around and lead time to market a product. For this design automation, system designers need modeling tools with high level of abstraction. With ever-increasing complexity of the processors and with growing number of special purpose processors, it has become even more important to have modular and generic modeling tools. In addition, the system designers need an integrated environment which allows them to assemble, compile, simulate and analyze the performance of various alternatives of the new design. Such an environment is essential to study the impact due to various software - hardware codesign trade-offs.

Toward this end, designers are using languages such as C, VHDL and Verilog. The difficulty in using these languages is that the level of abstraction provided by these languages doesn't allow rapid prototyping. Moreover, it is convenient to have one processor model for various applications such as simulation, assembler and disassembler generation and compiler back-end generation.

In this thesis, we have designed a language Sim-nML for processor modeling. The uniprocessor model developed with Sim-nML is helpful to generate processor specific tools such as assembler, disassembler, simulator etc., which form the core of design environment. We have also implemented an instruction set simulator generator. The generator takes Sim-nML specification as input and provides instruction set simulator in C as output. Sim-nML is primarily an extension of the nML[2] language.

1.1 Motivation

In accordance with Moore's law which states that processing power doubles ever 18 months, modern microprocessors are achieving phenomenal performance levels due to the advances in several enabling technologies. The increase of performance comes with greater complexity. The colossal complexity of modern microprocessors is attributed to various factors such as multiple functional units, out-of-order execution, branch prediction etc. The design of such complex superscalar processors requires the use of sophisticated software tools. Designers use functional and performance simulators to validate the functionality and to assess the performance of the processor for a given application. Typically, these tools are implemented using programming languages such as C, VHDL, etc., which requires enormous effort. Given an integrated environment in which it is easy to model uniprocessors this effort can be reduced significantly.

The complexity of microprocessors and the heavy market competition has led to several changes in system design process. The performance of the system not only depends on the microprocessor, but the external components such as caches, memory hierarchy etc. Nowadays, ad hoc system design techniques do not work because the success of the product depends on the performance of system with specific applications. A systematic design process starts with selecting the application and involves writing a model that measures performance of the system, testing the system, analyzing the results and refining the model to enhance performance. In this process, the model undergoes several changes till the desired performance is achieved. This systematic approach necessitates an environment which facilitates, incorporating the model changes and testing the model rapidly.

The systematic approach not only requires performance models, but also a set of tools which will facilitate compiling/assembling the applications. For example, if the instruction set architecture is under design and requires frequent changes then this necessitates reimplementing the assemblers and compilers. Therefore, it is desirable to have an environment where changes to the design are made at one place and the corresponding changes in other tools are automated. nML is an extensible machine description formalism and helpful to automate the generation of compiler, assembler,

disassembler and instruction set simulator from single specification. But, nML is not powerful enough to specify performance models due to lack of clear abstraction for specifying the control flow. This motivated us to extend nML to specify performance models so that it can be used to generate performance simulator in addition to the aforementioned tools.

1.2 Overview of Related Work

Automation tools for performance modeling of complete system is a growing area and enough research has been pursued in this area. These previous works has resulted in a set of performance modeling tools. Visualization Based Microarchitecture Workbench (VMW)[1] is an infrastructure which facilitates the specification of instruction set architecture and microarchitecture of a machine in concise manner. VMW provides a powerful environment for processor design. SimOS[3] project aims at providing a complete machine simulation environment which allows to run operating systems on simulated machine. This facilitates to analyze the performance under multiprogramming. Other than these complete machine simulation environments, many performance models exist for analyzing the individual components such as processors, caches etc. The processor performance model for Alpha processor is described in [6] and that for PowerPC is described in [5]. The cache performance models can be found in [9], [10]. In addition, there are herd of tools developed to speed-up the performance evaluation. This include execution profiling tools such as ATOM[7], Aint[4], etc. In Chapter 2, some of these works which are highly related to the work done in this thesis, are described..

1.3 Goals Achieved

In this thesis work, we aimed at developing an environment for performance modeling. The development of a complete integrated environment is in progress. The goals achieved in this thesis work are listed below.

- Resource Usage Model is a technique developed to abstract the control flow of instructions through pipelines.
- Sim-nML language is developed to specify the performance models with the help of resource usage model.
- Processor Instruction Set Simulator Generator is designed and implemented. The simulator generator takes Sim-nML specification as input and produces C++[8] code for simulator.
- A Hypothetical Superscalar Processor Model is specified in Sim-nML and the performance simulator is generated using simulator generator. This generated simulator runs at a speed of 4,000 instructions per second.

1.4 Organization of Report

The rest of the thesis is organized as follows. In Chapter 2 we give an overview of the performance models and examine some of the models designed in various projects worldwide. In Chapter 3, we describe the syntax and semantics of the Sim-nML. In Chapter 4 we describe the implementation of simulator generator which takes Sim-nML as input and produces C++ code for simulator. Finally we conclude in Chapter 5 and provide the results. We also enumerate possible future work in this area.

Chapter 2

Performance Analysis

In the early stage of the system design, executable specifications are converted to hardware and software components. In order to do so, the specifications are partitioned into two, what can be implemented in software, and, what needs to be implemented in the hardware. This objective of partitioning is essentially dependent on the processor used and the software, whether it can meet the real-time deadline or not. Such systematic approach to system design involves repeated performance testing and analysis. In this chapter, we discuss the salient features of modern microprocessors, their simulation techniques and survey some of existing performance analysis tools. We then provide a brief introduction to nML which has been extended to specify performance model and provide performance simulation capabilities in this thesis.

2.1 Salient Features of Modern Microprocessors

Modern microprocessors employ many performance enhancement techniques to boost their performance. However, inclusion of these techniques have increased the complexity of the microprocessor architecture substantially. While the general purpose microprocessors are reaching 600-MHz clock speeds, the number of transistors are touching 10 millions. On the other hand, the number of special purpose embedded system processors are increasing rapidly. With such a phenomenal hardware available

within a chip, many architectural techniques are being implemented in the hardware. Below, there is a list of some of architectural performance enhancement techniques used in high performance microprocessors today.

- **Pipelining** is the most basic technique used for performance enhancement. This facilitates execution of multiple instructions in overlapping fashion.
- **Multiple Functional Units** are helpful for executing more than one instruction simultaneously.
- **Branch Prediction** - Conditional branches are the main bottleneck for deep pipelines because target address for the next instruction fetch is not available till the branch is resolved. This has led to the development of branch prediction technique by which the target address is predicted early in the pipeline. In case of misprediction the pipeline is flushed out.
- **Register Renaming** is helpful to reduce the pipeline stalls due to data dependencies between instructions.
- **Feed Forwarding** is a technique by which a computed value is forwarded directly to other unit in the pipeline which requires it even before the value is written to the destination register.
- **Out-of-Order Execution** is a technique used to dynamically reschedule the instructions according to the availability of the resources. For example, if a floating point instruction follows an integer instruction and the integer instruction could not be issued because integer unit is busy, and the floating point unit is free then the floating point instruction will be issued before integer instruction.
- **Data Prediction** is a technique used normally to predict the branch target addresses in case of indirect branches where the target address is obtained from register. If the register containing the target address is being modified by some of the instruction down the pipe then the target address is predicted using either history or some clue from user.
- **Conditional instruction** is a technique used to reduce the number of conditional branches. In most of the cases, the body of conditional branches contain

only very few instructions. Under this technique, the instructions in the body of the conditional branch, are converted to conditional instructions and the conditional branch is removed. A conditional instruction is executed even though the value of the condition flag is not known, but the result is written if-and-only-if the condition is true. If condition turns out to be false, then the effort for executing the instruction goes waste. But, the effort loss is very less compared to the pipeline flush in case conditional branch is used and the target is mispredicted.

2.2 Simulation Methods

The colossal complexity of modern processors due to the aforementioned performance enhancement techniques, makes the simulation a time consuming process. However, it is essential to develop a fast simulator which will be just order of magnitude slower than the original processor. This is not a simple task to achieve and many techniques are developed to speed-up the simulators. Below, we classify the simulation methods according to the technique employed.

2.2.1 Depending on Resource Management

■ *Cycle Based Simulation*

In cycle based simulation, a simulator clock is maintained which is analogous to the processor clock. In each simulation cycle, all active instructions, are simulated serially. For each instruction the availability of the next set of resources such as the next stage in the pipeline, is explored. If resources are available then the instruction is marked active for the next cycle. This process stops when a resource becomes unavailable for the next cycle or when the instruction is simulated completely to the end. The main disadvantage of this approach is that considerable effort is put in each simulation cycle for checking the availability of the next set of resources used by an instruction when they are not available. The major advantage of this method is the simplicity in implementation.

■ *Event Based Simulation*

This method overcomes the aforementioned disadvantage of cycle based simulation by maintaining a list of instructions waiting for a particular resource. The waiting instruction is resumed for simulation as soon as the resource becomes free. Thus, at each simulation cycle the checking of enabled instructions is eliminated. However, this method requires an additional complexity of maintaining a list of waiting instructions. It is apparent that if instructions wait frequently for the resources for more than one clock cycle, then this method outperforms cycle based simulation.

2.2.2 Depending on Instruction Sequence

■ *Execution Driven Simulation*

In this method, the actual functionality of the instruction is simulated in addition to the simulation of flow of instructions through the processor pipelines. The dynamic sequence of instructions is obtained by resolving branches with the help of user inputs. The main disadvantage of this approach is that this simulation is slow. The main advantage of this approach is that the model is tested for its functionality as well as its performance.

■ *Trace Driven Simulation*

In this method, developers run an instrumented version of the program with the help of a simple instruction set simulator[4] (without pipelining and timings) to obtain a time sequence of instructions. If a prototype hardware already exists, then the time sequence of instructions can be obtained by running the instrumented version on the prototype[6]. For example, 80486 can be used as a prototype for Pentium, This time sequence is referred to as trace which is fed to a fast simulator which only models the pipeline flow. These simulators are relatively fast because they do not emulate the actual functionality. Stand-alone trace analysis tools are available for generation of traces and these facilitate fast performance analysis. The simulation done after trace generation is called Trace driven simulation. The main disadvantage of this approach

is that it is hard to validate the model as results are not produced.

2.3 A survey of Performance Analysis Tools

2.3.1 Alpha Microprocessor Model

Digital Equipment developed a performance model[6] for evaluating the Alpha processor. In this model, the runtime traces are collected with the help of a tool called Atom[7]. Then these traces are fed to a performance modeling tool which is implemented in about 50,000 lines of C code. The performance modeling tool simulates about 10,000 to 20,000 instruction traces per second. The performance modeling tool provides logging facility and graphical interface to view the outputs.

2.3.2 SimOS

SimOS[3] is an ambitious project with the objective of simulating the complete hardware in enough detail to run system software including commercial operating systems. In SimOS, each hardware component is modeled by a set of simulation models which differ in their timing accuracy and speed. SimOS allows to interchange simulation models at runtime. The current version of SimOS has two performance simulation models, viz., Mipsy and MXS for MIPS CPU. Mipsy is a simple pipeline model and MXS is complex dynamically scheduled processor model. MXS can only simulate on the order of 20,000 instructions per second. Because of this slow simulation speed, Mipsy, which is an order of magnitude faster is used to warm up the caches before switching into MXS.

2.3.3 VMW

VMW[1] is a visualization based microarchitecture workbench developed in Carnegie Mellon University. In VMW, a machine is specified by five different specification files. These files describe the syntax, semantics and timing of the instructions and the

microarchitecture of the processor. Using these specification files and the VMW infrastructure a performance simulator can be generated. The VMW infrastructure is implemented in C++ as a class structure hierarchy and supports trace driven simulation. The simulation is performed by executing the machine behavior code specified by the user. This machine behavior code describes the control logic used to process the instructions through the pipeline stages. The machine behavior code interacts with the VMW infrastructure class hierarchy for extracting the syntax and semantic information from other specification files, managing long traces, managing the resources etc. VMW infrastructure provides extensive visualization tools for user interface.

2.3.4 nML

nML [2] is an extensible formalism targeted for describing arbitrary single processor computer architecture. nML works at instruction set level and hides the implementation details. In nML, the instruction set is enumerated by an *attribute grammar*¹. The semantic action of any instruction is composed of fragments that are distributed over the whole specification tree, i.e., the common behavior of a class of instructions is captured at the top level of the tree and the specialized behavior of sub-classes are captured in the subsequent lower levels.

■ nML Grammar

nML grammar has a fixed start symbol namely *instruction* and two kind of productions namely, *or-rule* which looks like,

$$\text{op } n0 = n1 \mid n2 \mid n3 \mid \dots$$

and *and-rule* which looks like,

¹An attribute grammar is a context free grammar in which for each non-terminal a fixed set of attributes and for each production a set of semantic rule is given. In nML grammar, all non-terminals have to have derivations. So, we don't differentiate between productions and non-terminals.

```

op n0 ( p1 : t1, p2 : t2, ... )
a1 = e1  a2 = e2 ...

```

where each n_i is a non-terminal and each t_i is a token. Each a_i is an attribute name and e_i their respective definition. The p_i are names of the parameters used in the attribute definitions.

nML grammar pre-defines three attributes namely *syntax*, *image* and *action*. The *syntax* attribute describes the textual syntax of the instruction. The *image* attribute describes the binary coding of the instruction and *action* attribute describes the semantics of an instruction.

The nML description in Figure 1, is that of a simple machine with two instructions, the add instruction which is used to add an argument to the accumulator AC and, the multiply instruction which is used to multiply the accumulator AC to the argument. The register PC has special semantics and points to the next-to-be-executed instruction.

In most of the processors, addressing modes and instructions are orthogonal to each other. Therefore, describing an instruction with each of the possible addressing modes explode the size of the description. Therefore, nML separates addressing mode description. For example, register addressing mode can be described as shown in Figure 2.

In addition, nML supports macros and declarations for types and constants. This enhances the clarity of the description.

nML formalism helps in describing the processor concisely and precisely. nML description of a processor can be used as input to various tools such as assembler and disassembler generators, compiler back-end generators and general purpose instruction set simulators. However, nML lacks control flow constructs and cannot be used for describing the inter-instruction dependencies. Further, it is not possible to specify the timing for various operations. Therefore, it is not possible to use nML for performance estimation.

```

type addr = card ( 32 )
type byte = card ( 8 )

mem AC [ 1, byte ]
mem PC [ 1, addr ]

op plus ( ) .
syntax = "add"
image = "000000"
action = { AC = AC + tmp; }

op multiply ( )
syntax = "mult"
image = "000001"
action = { AC = AC * tmp; }

op binaction = plus | multiply

op instruction ( x : binaction, data : byte )
syntax = format ( "%s %d", x.syntax, data )
image = format ( "11%6b %8b", x.image, data )
action = {
    PC = PC + 2;
    tmp = data;
    x.action;
}

```

Figure 1: nML Specification for a Simple Processor

```

mode REG ( n : card ( 5 ) ) = A [ n ]
syntax = format ( "%d", n )
image = format ( "%5b", n )

```

Figure 2: Register addressing Mode Specification

Chapter 3

Sim-nML Syntax and Semantics

In this chapter, we describe Sim-nML language which can be used for high level performance modeling. We first describe the resource usage model which is the basic philosophy behind the design of Sim-nML. We then describe the syntax and semantics of *uses* attribute, which helps to specify resource usage model. Finally we conclude this chapter with specification of a simple superscalar processor and branch prediction, in Sim-nML.

3.1 Resource Usage Model

As seen earlier, nML formalism is not useful for performance modeling. This is mainly because of the lack of a mechanism to specify the control flow. Therefore, we extend nML, by abstracting out the control flow with the help of resource usage model.

The resource usage model is based on the fact that at any instant, an instruction in execution, holds a set of resources and performs some action. The resources held by the instruction and the action taken change progressively, in time.

In resource usage model, a resource is an abstraction of a piece of hardware such as a register, ALU, a functional block, etc. for which instructions contend. The control flow is simply a way of resolving conflicts due to contention. In our model, we can use one of the two methods to specify control flow. The first method is to

specify the time units for which each of the acquired resource is used and the second method is to specify a condition which should be true to proceed further. This model resembles the actual behavior of microarchitectures and facilitates the specification of microarchitecture of the processor at a higher level of abstraction.

For example, consider our simple processor described in the Chapter 2. We model the processor with three pipeline stages, viz., `fetch_unit`, `execution_unit` and `retire_unit`. The Sim-nML specification of the processor is given in Figure 3. It specifies that all instructions first use the `fetch_unit` for one unit of time. The instructions then use the `execution_unit` for the duration dependent on the instruction and then the `retire_unit` for one unit of time. The `add` instruction uses the `execution_unit` for one time unit whereas the `multiply` instruction uses the `execution_unit` for three time units. The `token action` at the end of `uses` specifies that after the specified resources are used for the specified duration, the function specified in `action` attribute is performed. The `resources` declaration is used to declare the functional blocks such as the `fetch_unit`, the `execution_unit` and the `retire_unit`. The description of actual functionality of these resources is not in the scope of Sim-nML formalism and is hidden. The declaration `reg` is same as `mem` declaration and is described in Section 3.2.7

The primary extension made to incorporate resource usage model in nML is, the addition of a new pre-defined attribute `uses`. The `uses` attribute is used to describe the resource usage model and the control flow of an instruction. The exact syntax is shown in Section 3.2.1.

The unit of time can be thought-of as similar to the machine clock cycle although it is not a restriction imposed by Sim-nML. However, sub-unit timings are not allowed. In a nut-shell, if unit of time is same as machine clock cycles then we can estimate the number of clock cycles taken by a program to complete.

```

resource fetch_unit, execution_unit, retire_unit

type addr = card ( 32 )
type byte = card ( 8 )

reg AC [ 1, byte ]
reg PC [ 1, addr ]

op plus ( )
syntax = "add"
image = "000000"
action = { AC = AC + tmp; }
uses = execution_unit #1

op multiply ( )
syntax = "mult"
image = "000001"
action = { AC = AC * tmp; }
uses = execution_unit #3

op binaction = plus | multiply

op instruction ( x : binaction, data : byte )
syntax = format ( "%s %d", x.syntax, data )
image = format ( "11%6b %8b", x.image, data )
action = { tmp = data; x.action; }
preact = { PC = PC + 2; }
uses = fetch_unit : preact & #1, x.uses, retire_unit #1 : action

```

Figure 3: Sim-nML Description of the Simple Processor

3.2 Specification of resource usage model using uses attribute

3.2.1 Uses Grammar

The *uses* attribute is the key construct in describing the resource usage model. The context free grammar for uses definition is given Figure 4 and 5.

```

uses-attr:
    uses = uses-def

uses-def:
    uses-or-sequence
    uses-def , uses-or-sequence

uses-or-sequence:
    uses-if-atom
    uses-or-sequence | uses-if-atom

uses-if-atom:
    uses-indirect-atom
    if boolean-expr then uses-indirect-atom
    if boolean-expr then uses-indirect-atom
    else uses-indirect-atom

uses-indirect-atom:
    uses-and-atom
    token . uses
    ( uses-def )

uses-and-atom:
    uses-cond-def-atom
    uses-action-atom
    uses-action-atom & # time

uses-action-atom:
    uses-cond-def-atom : token
    uses-cond-def-atom : token . token

uses-cond-def-atom:
    uses-def-atom
    uses-cond-atom

```

Figure 4: uses Grammar

```

uses-def-atom:
    # time
    token # time
    token-and-list
    token-and-list & # time

token-and-list:
    token
    token-and-list & token

uses-cond-atom:
    token relop token
    token relop constant
    " token " ( token-list )

token-list:
    token-list-part
    token-list , token-list-part

token-list-part:
    token
    token . uses

relop: one of
    ==, <, >, <=, >=, !=

```

Figure 5: uses Grammar (Cont.)

Although the grammar seems to be very complex and very restrictive, it is simple and powerful enough to specify the resource usage model of modern processors. The grammar is designed carefully to avoid semantic gaps as much as possible. There are some compromises made in terms of length of the grammar for attaining semantic precision.

In resource usage model, a set of resources are acquired by an instruction and the resources are held till the next set of resources are available. When the first instruction after the pipeline flush enters the pipeline all the resources are immediately available. Therefore, to control the flow of instructions, it is also necessary to specify the

time for which each resource is held. For example, `fetch_unit #1, execution_unit #1, retire_unit #1` means that at first the `fetch_unit` is acquired. Although, `execution_unit` is available immediately, the instruction waits in `fetch_unit` for one time unit before acquiring the `execution_unit`. Then, it holds `execution_unit` for one time unit. Before completion, it acquires `retire_unit` and holds it for one time unit. In some cases, it is not possible to specify the resource hold times statically. The instruction has to wait till a condition becomes true. For example, to model in-order-retirement of instructions, an instruction before completion should wait till the completion of all the instructions that precede it. For more explanation see Figure 7.

In the grammar shown in Figure 4, `uses-cond-def-atom` describes a set of resources and the time duration for which these resources are used or a condition for which the instruction waits to become true. Hereafter, we use the term *resources* to represent the resources which an instruction needs to proceed further as well as the condition for which an instruction has to wait. The comma (,) signifies the sequencing of the usage of resources. For correct implementation of the resource usage model semantics that an instruction holds a set of resources at any instant, it should be made sure that the next set of resources is available before freeing the already held resources.

The `and (&)` operator is used to represent a set of resources all of which are used simultaneously. For example, `execution_unit & AC & #1` means that the instruction needs to acquire `execution_unit` and `AC` and having acquired that, it holds on to them for 3 time units. The syntax "`token #time`" is same as "`token & #time`". Different alternatives are provided to increase the readability of description. The grammar does not allow specifying more than one hold time for the same set of resources. For example, it is not possible to specify `execution_unit & #3 & AC & #1`. This specification is ambiguous because it is not clear whether the resources are held for 3 time units or one time unit.

The `colon (:)` operator is used to specify the instant at which an action should be carried out. An instant in resource usage model is relative to the time of acquiring a set of resources. For example, `fetch_unit : preact` means that the function specified in `preact` attribute should be performed after acquiring `fetch_unit` and `retire_unit #1 : action` means that the function specified in `action` attribute

should be performed one time unit after acquiring `retire_unit`.

The `or (||)` operator signifies that any one of the resources is used. If two or more resource sequences are `ored` then the resource at the head of each sequence is used for resolving the conflict. For example, `uses = (pipe11, pipe12) || (pipe21, pipe22)`, means that if `pipe11` is free then the first sequence is selected as the `uses` sequence, otherwise if `pipe21` is free then second sequence is selected as the `uses` sequence. If both are not free, then the instruction is blocked till one of them becomes free. This is useful for selecting the free unit from a set of identical functional units.

The `uses-if-atom` facilitates selection of a sequence of resources from two different alternatives depending on the truth value of the boolean expression. The syntax `"token" (...)` signifies a canonical function call¹. The indirection construct `token.uses`, is useful for describing resource usage model in a hierarchical manner. For example, in Figure 3 the integer addition and integer multiplication instructions which use same sequence of resources except that multiplication consumes more time, is specified hierarchically. With the above `uses` syntax, we can specify the resource usage model concisely and precisely.

3.2.2 Implicit Parallelism

At instruction level, the processors are inherently parallel in the sense that the instruction execution is overlapped either with the execution of other instruction or with other functionalities such as checking for interrupt etc. The instruction-by-instruction specification of the behavior in `Sim-nML` is powerful enough to model a complex processor design with high degree of parallelism. Simulator generators can use the `Sim-nML` specification to generate a simulator that can simulate the parallel instruction execution. The resource usage maps described in the instruction specification streamlines the execution of the instructions in the desired way. In `Sim-nML`, if two instructions require a set of resources simultaneously, then the conflict is resolved by following strict FIFO order i.e. the earlier instruction is allocated the set of

¹Canonical functions[2] are those functions whose semantics are known only to the entity that reads the description. In our `Sim-nML` implementation, other than few pre-defined canonical functions, all other canonical functions are mapped directly to C++ functions.

resources and the later instruction waits. However, this default FIFO behavior can be overridden by explicit specification with the help of condition waits.

We show the power of the *uses* attribute with a description for a simple superscalar processor with two integer units. The block diagram of the processor is shown in Figure 6. The instruction set architecture of this superscalar processor is similar to the one in the earlier example. In a superscalar design, we encounter data dependency hazards. So, our design should take care of these dependency hazards. This is done by properly specifying the accumulator AC usage in the *uses* attribute.

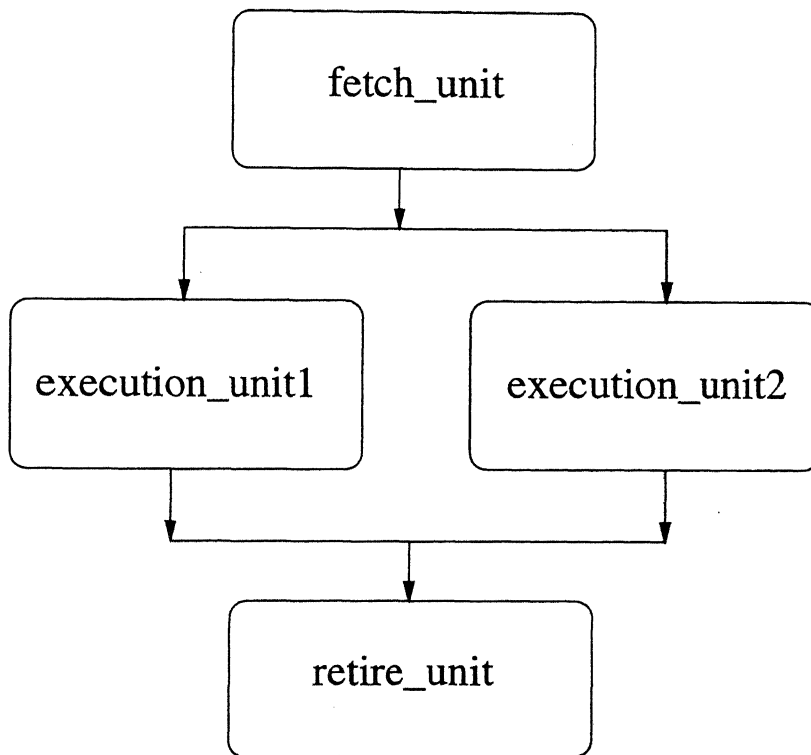


Figure 6: Block Diagram of A Superscalar Processor

The Sim-nML description in Figure 7 models this hypothetical superscalar processor with two identical execution units. In this processor, the instructions are issued to the execution units alternatively. Therefore, the **retire_reg** is set to the value 1 and 2 alternatively. This leads to retiring of instructions in order. The action attribute of **initial** specifies the initial values of the registers.

```

resource fetch_unit, execution_unit1,
           execution_unit2, retire_unit
reg AC [ 1, card ( 8 ) ]
reg PC [ 1, card ( 32 ) ]
reg ireg [ 1, card ( 8 ) ]
reg retire_reg [ 1, card ( 8 ) ]

op plus ( data : card ( 8 ) )
syntax = format ( "add %d", data )
image = format ( "000000%8b", data )
action = { AC = AC + data; }
uses = ( ireg == 1, execution_unit1 & AC & #1, retire_reg == 1 ) |
        ( ireg == 2, execution_unit2 & AC & #1, retire_reg == 2 )
op multiply ( data : card ( 8 ) )
syntax = format ( "mult %d", data )
image = format ( "000001%8b", data )
action = { AC = AC * data; }
uses = ( ireg == 1, execution_unit1 & AC & #3, retire_reg == 1 ) |
        ( ireg == 2, execution_unit2 & AC & #3, retire_reg == 2 )

op binaction = plus | multiply
op initial ( )
action = { issue_reg = 2; retire_reg = 1; }

op instruction ( x : binaction )
syntax = format ( "%s", x.syntax )
image = format ( "11%6b", x.image )
action = {
    x.action;
    if ( retire_reg == 1 )
        then retire_reg = 2; else retire_reg = 1; endif;
}
preact = {
    PC = PC + 2;
    if ( ireg == 1 )
        then ireg = 2; else ireg = 1; endif;
}
uses = fetch_unit : preact & #1, x.uses,
       retire_unit & AC & #1 : action

```

Figure 7: Sim-nML Description of a Superscalar Processor

3.2.3 Specification of Feed Forwarding

In processors supporting feed forwarding mechanism, data is forwarded as soon as it is available to reduce the pipeline stalls due to data dependencies. In case of software timing estimation, where the processor is specified as in Figure 7, it is easy to specify feed forwarding by adjusting the register blocking time appropriately.

3.2.4 Specification of Memory Hierarchy

To avoid the bottleneck on system performance imposed by high memory access latency, modern machines use hierarchical memory. The precision of handling memory hierarchies depends on the specification. In Figure 8, we model a memory system with a `data_cache` whose hit ratio is presumed to be 95%. The time to access data from `data_cache` is assumed to be two units of time and for that from `main_memory` is assumed to be ten units of time.

```
mem data_cache [ 1024, word ]
uses = #2

mem main_memory [ 2**16, word ]
uses = #10

//Register Indexed Address Mode.

mode IND ( R : Address_Register, x : mem_access ) = M [ R ]
syntax = format ( "(A%3b)", R )
image   = format ( "#3b", R )
uses    = if "drand48"() < 0.95 then data_cache.uses
          else main_memory.uses
```

Figure 8: Specification of Memory Hierarchy

In Figure 8, "drand48" denotes a canonical function call. A more precise cache model can be built, for example, by writing a C++ function which keeps track of the contents of the cache, as shown in Figure 9. In this model, the `is_hit` function can implement the cache replacement policy such as least recently used or less frequently

```

mode IND ( R : Address_Register, x : mem_access ) = M [ R ]
syntax = format ( "(A%3b)", R )
image  = format ( "#3b", R )
uses   = if "is_hit"( data_cache, R )
          then data_cache.uses else main_memory.uses

```

Figure 9: Precise Specification of Memory Hierarchy

used etc., using the address argument. The above example shows the ease with which the semantics of description can be increased. The implementation details are discussed in the next chapter.

3.2.5 Exception Declaration

For an accurate performance modeling and for specifying inter-instruction dependencies, it is important to handle the exceptional conditions such as divide-by-zero, interrupts and branch-prediction-error. In Sim-nML language certain pre-defined canonical functions are added to handle exceptions. The `exception` declares the different kinds of exceptions. Handling exceptional conditions is complicated because it is likely that

```

exception divide-by-zero-error, branch-error, interrupt

```

the exceptions are handled differently at different machine states. For example, when a branch-prediction-error occurs the instructions along the mispredicted path should be flushed out of the pipeline. To simplify the design, in modern processors, these instructions are either executed to completion without writing the result to the registers if the instruction is already issued to execution unit or flushed out and the resources held by the instruction are freed immediately if the instruction is not issued. Therefore, it is necessary to provide mechanism to change the handler at any instant. In our model, we propose the following canonical functions related to the exceptions.

- **sethandler** - used to set a new handler for an exception

- `ignore` - used to specify that an exception can be ignored
- `raise` - used to broadcast the occurrence of an exception to all instructions in execution.
- `abort` - used to abort an instruction on execution and free the resources held by the instruction immediately.

3.2.6 Instruction Identification

While specifying a superscalar processor, it is necessary to identify every instruction issued to streamline the retirement of instructions. In case of the simple hypothetical superscalar processor described in the Figure 7, retirement is streamlined by alternately assigning `retire_reg` to 1 and 2. This serves the purpose of identifying the instructions. However, for complex processors, instruction identification becomes difficult to specify. To facilitate this task, we have introduced a type `instid_type`, which is opaque to the designer. It can only be told that the result of any operation on this type is modulo maximum number of instructions that can be active inside the processor at any time. The code for implementing the type is generated automatically. The designer can help the `instid_type` code generation, by giving a clue about the maximum number of instructions that can be active inside the processor at any instant. This can be done by defining a reserved constant `MAX_INSTR_COUNT`. The value of instruction identifier can be obtained through a canonical function `"instid"()`.

3.2.7 Initial State and `reg` Declaration

Since Sim-nML is used for generating instruction set simulator, the initial state of the processor has to be specified. This is done with a reserved op, named as `initial`. The `action` attribute of `initial`, describes the initial state of the processor. In simulation, it is better to distinguish main memory from internal processor registers. To facilitate this we have introduced `reg` declaration which declares a memory location as a register. The normal `mem` declaration specifies memory locations which are not registers. The main difference is that each component of a `reg` declaration is con-

```
mem M [ 2**32, word ]  
reg R [ 32, word ]
```

Figure 10: reg declaration

sidered as a resource whereas all the locations declared with `mem`, are grouped as a single resource. In Figure 10, `M[2000]` and `M[3000]` are considered as a part of single resource `M`, whereas `R[1]` and `R[2]` are considered as different resources. There may be other difference such as space allocation, which are dependent on implementation of tools which use Sim-nML specification.

3.3 Specification of Superscalar Processor and Branch Prediction

Most of the modern processors use branch prediction for increasing the performance. There are various branch prediction policies such as branch always taken, static branch prediction, dynamic branch prediction, etc. Abstracting out all such diversification complicates the grammar and the semantics. But, with the help of constructs available and minimal implementation detail it is easy to specify the branch prediction mechanism as shown in this section.

In Figure 11, the block diagram is given for the superscalar processor being considered. The processor supports branch prediction. The processor contains an accumulator and a count register and a zero flag register `Z`, which is set only if the result of computation is zero. The Sim-nML code in Figure 12, describes the decrement instruction which decrements the count register and plus instruction which adds an immediate value to the accumulator. The code in Figure 13, describes the multiplication instruction. The condition wait in `non_branch_instr`'s uses, fashions in order retirement of non branch instructions. In case of branch misprediction, the value of `reorder_buffer` associated with a particular non-branch-instruction is set to 255. Therefore, the action should be carried out if-and-only-if the value is not 255. Figure 13 and 14 describe the branch instruction. The processor follows always taken policy

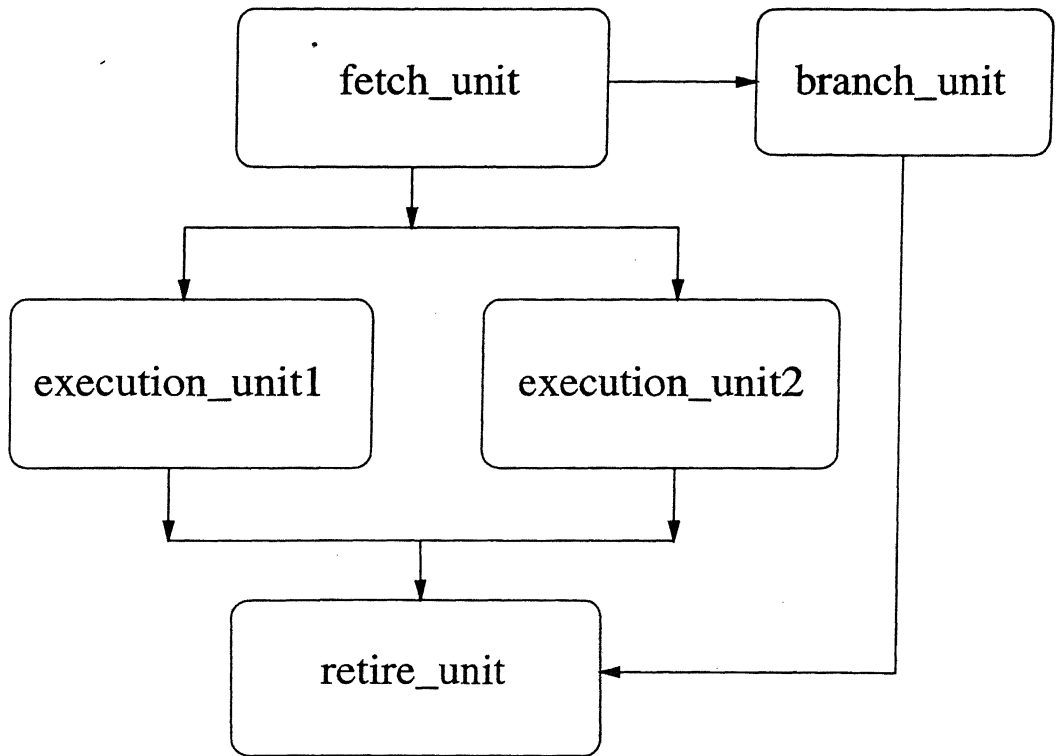


Figure 11: Block Diagram of A Superscalar Processor With Branch Unit

for branch prediction. In case of speculation, the value of Z is checked once it is evaluated. If prediction is found to be wrong then a `branch_error` is signaled which is caught by all instructions. On catching the `branch_error`, the instructions that follow branch instruction set their associated value in `reorder_buffer` to 255. Since this model allows only one branch instruction at any instant, nested branch errors are not taken care of.

```

const MAX_INSTR_COUNT = 8
resource fetch_unit, execution_unit1, execution_unit2,
        retire_unit, branch_unit, halt_unit
exception branch_error
type addr = card ( 32 )
type byte = card ( 8 )
type bit  = card ( 1 )
mem AC [ 1, byte ]
mem PC [ 1, addr ]
mem Z  [ 1, bit ]
mem Count [ 1, byte ]
mem speculated [ 1, bit ]
mem oldpc [ 1, addr ]
mem retire_reg [ 1, instid_type ]
mem reorder_buffer [ MAX_INSTR_COUNT, byte ]

op initial ( )
action = { PC = 0; Count = 10; }
op dec_count ( )
syntax = "dec_count"
image  = "00001000000000000000000000000000"
action = {
        Count = Count - 1;
        if Count == 0
            then Z = 1; else Z = 0; endif;
}
uses    = execution_unit1 #1 | execution_unit2 #1

op plus_inst ( data : byte )
syntax = format ( "add %d", data )
image  = format ( "0000000%24b", data )
action = {
        AC = AC + data;
        if AC == 0
            then Z = 1; else Z = 0; endif;
}
uses    = execution_unit1 #1 | execution_unit2 #1

```

Figure 12: A Superscalar Processor with Branch Prediction

```

op mult_instr ( data : byte )
syntax = format ( "mult %d", data )
image = format ( "000001%24b", data )
action = {
    AC = AC * data;
    if AC == 0
        then Z = 1; else Z = 0; endif;
}
uses = execution_unit1 #3 | execution_unit2 #3

op binaction = plus_inst | mult_instr | dec_count

op non_branch_instr ( x : binaction )
syntax = format ( "%s", x.syntax )
image = format ( "11%s", x.image )
action = { x.action; }
uses = fetch_unit #1, x.uses,
    retire_reg == "instid()", retire_unit & AC & Z & #1,
    if reorder_buffer [ retire_reg ] != 255
        then retire_unit : action

op branch_instr ( target : addr )
syntax = format ( "brnz %24b" , target )
image = format ( "01000000%24b" , target )
preact = {
    bran_inst_id = "instid()";
    reorder_buffer [ bran_inst_id ] = 255;
}
action = {
    if Z == 0
        then PC = target; endif;
    speculated = 0;
}
always_taken = {
    oldpc = PC;
    PC = target;
    speculated = 1;
}

```

Figure 13: A Superscalar Processor with Branch Prediction (Cont.)

```

restore = { bran_inst_id = 88888; }
check_prediction = {
    if Z == 1 then
        PC = oldpc;
        "raise" ( branch_error );
    endif;
}
uses = if "is_blocked" ( Z )
    then fetch_unit : always_taken & #1
    else fetch_unit : action & #1,
    branch_unit : preact,
    if speculated == 1 then Z : check_prediction,
    retire_reg == bran_inst_id, retire_unit #1 : restore

op instr_action = branch_instr | non_branch_instr

op instruction ( x : instr_action )
syntax = x.syntax
image = format ( "%s", x.image )
action = { x.action; }
handler= {
    if reorder_buffer [ "instid"() ] == bran_inst_id
    then reorder_buffer [ "instid"() ] = 255; endif;
}
preact = {
    PC = PC + 4;
    reorder_buffer [ "instid"() ] = bran_inst_id;
    "sethandler" ( branch_error, handler );
}
reorder = { retire_reg = retire_reg + 1; }
uses = fetch_unit : preact , x.uses, retire_unit : reorder

```

Figure 14: A Superscalar Processor with Branch Prediction (Cont.)

Chapter 4

Implementation

In this chapter, we discuss the different design alternatives for implementing performance simulators and the associated trade-offs. We then describe the various phases in the implementation of instruction set simulator (ISS) generator. The earlier phases are helpful in collecting the hierarchical information entrenched in the specification. The last phase generates C++ code for simulator. Finally, the implementation of nML type system is described.

4.1 Design of Simulator

Our objective is to develop a ISS generator which generates a simulator for a processor from its Sim-nML description. The simulator generator produces C++ code of the simulator. Before, studying the design of simulator generator, it is important to know the output produced by it. Therefore, in this section we concentrate on the design of the simulator that is generated.

The ISS generator outputs a C++ code for the simulator. We choose C++ because of its extensible type system which facilitates implementation of nML types such as `card(1)`, `card(2)` etc. Moreover, this choice lends us greater flexibility in linking already developed performance models such as efficient cache models, to generated processor model.

There are a herd of other issues which should be considered in simulator design. These issues arise from the resource usage model, complex multidimensional parallelism of processor, maintaining the state of the processor, and the strict typing in C++. Below we describe the issues in detail.

- Resource usage model is based on the fact that instructions hold a set of resources at any instant of time. Therefore, it should be made sure that the next set of resources are available before releasing the already held resources. Moreover, when an instruction is waiting for a sequence of conditions (or clocks), the present set of resources are held.
- Sim-nML is inherently parallel language and the instructions are modeled to execute in parallel in case the required resources are available. Therefore, the simulator has to store the context of the present instruction and resume simulating other instructions if a particular instruction simulation is stalled by the non-availability of required resources.
- Inherent parallelism results in higher complexity in abstracting out the state of the processor. The state of processor in the context of the simulator is defined by the register and memory values and the state of the partially simulated instructions. The implementation of the maintenance of state of the partially simulated instructions is hard. Therefore, we concentrate on this issue in Section 4.1.1.
- Although, extensible type system of C++ helps in implementing new types, its rigidity results in difficulties in storing the state of the partially simulated instructions and dynamically invoking the functions specified in Sim-nML's function attributes such as action.

4.1.1 Capturing Machine State

From the arguments described above, it is clear that the capturing the state of the machine constitutes the major part of simulator design. With the issues in designing a simulator for Sim-nML specification well understood, we propose two designs for capturing the machine state.

■ Design - 1

In this design, the full context of an instruction is abstracted as a sequence of resource requests and corresponding actions to be taken on acquiring these resources. We have designed a basic data structure for storing a component of the sequence. We call this data structure as `uses_node` and a simplified version of the structure is shown in Figure 15. The `node_type` signifies the different kinds of the node. Some of the

```
int  node_type;
int  wait_resource [ MAX_NO_OF_RESOURCES ];
int  free_resource [ MAX_NO_OF_RESOURCES ];
bool (*condition_wait)();
bool (*if_condition)();
void (*action)();
int  pre_time;
int  post_time;
int  jump;
```

Figure 15: `uses_node` Structure

possible values of `node_types` are `SIMPLE`, `IF`, etc. Each resource is identified by a unique number. The identifiers of resources for which an instruction waits at any time instant are stored in an array called `wait_resource`. When an instruction moves from one state to other it frees the previously held resources and holds new set of resources. However, in some cases, only some of the the previously held resources are freed. In order to implement such behavior, we store the resources to be freed separately in an array called `free_resource`. The conditional waits are implemented by encapsulating the condition through a boolean function and storing the function pointer in `condition_wait` variable. The action specifies the action to be carried out at this instant. In case of `SIMPLE` node, the simulator waits for `condition_wait` function to evaluate to true and then acquires resources specified in `wait_resource`. Then, it waits for `pre_time` time units to execute action. After action is executed, simulator waits for `post_time` time units before freeing the resources specified in `free_resource`. To correctly implement resource usage model, the simulator finds the next appropriate `uses_node` and makes sure that `condition_wait` of that node

evaluates to true and the resources specified in that node are available, before freeing the present resources.

Finding next appropriate `uses_node` in the sequence, depends on the type of the successor node. There are five node types viz. `SIMPLE`, `IF`, `JUMP`, `OR`, `END_OR`. The next appropriate node is found with help of the algorithm shown in Figure 16. In

```
int find_next ( int successor )
{
    switch ( node_type ( successor ) ) {
        case SIMPLE:
            return successor;
        case JUMP:
            return find_next ( successor + jump );
        case IF:
            if ( if_condition() )
                return successor;
            else return find_next ( successor + jump );
    }
}
```

Figure 16: Finding next appropriate `uses_node`

case successor node type is `OR`, then the node is checked for readiness i.e. whether `condition_wait` evaluates to true and whether resources specified in `wait_resource` are available. If successor node is ready then it is considered as next appropriate node. Otherwise, the node numbered `successor + jump` is checked for readiness. The node type `END_OR`, denotes the end of `ored` subsequences. If all the nodes till `END_OR` are not ready, then simulator continues to wait in the present state.

Simulator keeps a list of instructions and when one instruction gets blocked due to one or more of the aforementioned reasons, the simulator stores the partially simulated context by storing the present node number in the sequence. Then it resumes the simulation of the next instruction. The present implementation is clock based and resumes with next instruction even though it is blocked.

■ Design - 2

In this design, each and every state of a partially simulated instruction is stored in a sequence of integer variables. On resuming the instruction for simulation, these numbers are used to make the choice in the nested switch statements. The total context of the instruction is described by nested switch statements generated by the simulator generator and the sequence of numbers used to restore the context. The classes generated for the simple Sim-nML description in Figure 3, chapter 3, looks like the code in Figure 17. In the code shown in the Figure 17, the `resource_id` function returns an integer which serves as the identifier for the resource whose pointer is used as the argument on calling the function. The variable `state` in each class stores the state information required to restore the context.

The `uses` function returns three kind of values viz. `START`, `WAIT` and `FINISH`. The value `WAIT` signifies that the instruction is waiting for some resource or clock and `FINISH` signifies that the `uses` function has finished. The significance of `START` is discussed below. Implementing `if` in Sim-nML *uses*, is straight forward. This can be done as shown in Figure 18.

Implementation of `or` in Sim-nML *uses*, is similar to the implementation of `if`. In case of `or`, the first set of resources are checked for availability. If resources are available, then the selection between different alternatives is resolved. In the case of redirection i.e. `x.uses()`, if the return value of new `uses` function is `START` then this signifies that the first resource is not available. If resources are not available, then the state variable is set to the start of next alternative and loop is repeated with help of `break` statement. This is repeated till all the alternatives are exhausted. If all alternatives fail then the state is reset to the starting of the `or` and `WAIT` is returned.

4.1.2 Capturing Hierarchical Structure

nML has demonstrated the power of specifying the instruction set architecture hierarchically. The hierarchical structuring helps to specify the Resource Usage Model in a concise and precise manner. Therefore, hierarchy is maintained in Sim-nML. However, at some point of time, the hierarchy should be collapsed to produce flat structured

```

class plus {
    int state;
    ...
    int uses ( ) {
        int wait_resource [ MAX_NO_OF_RESOURCES ];
        switch ( state ) {
            case 0:
                wait_resource [ 0 ] = resource_id( &execution_unit );
                if ( ! block ( wait_resource, 1 ) ) return START;
                state++;
            case 1:
                if ( ! clock ( 1 ) ) return WAIT;
                state++;
            case 2:
                return FINISHED;
        }
    }
};

class instruction {
    int state;
    ...
    int uses ( ) {
        int wait_resource [ MAX_NO_OF_RESOURCES ];
        switch ( state ) {
            case 0:
                wait_resource [ 0 ] = resource_id ( &fetch_unit );
                if ( ! block ( wait_resource, 1 ) ) return START;
                preact();
                state++;
            case 1:
                if ( ! clock ( 1 ) ) return WAIT;
                state++;
            case 2:
                if ( x.uses ( ) != FINISHED ) return WAIT;
                state++;
            ...
        }
    }
};

```

Figure 17: Using Switch Statements to Restore Context

```

int uses ( )
{
    while ( true ) {
        switch ( state ) {
            case 0:
                if ( retire_reg == 1 )
                    state++;
                else {
                    state = 3;
                    break;
                }
            case 1: //if part
                ...
            case 2:
                ...
            case 3: //else part
                ...
        }
    }
}

```

Figure 18: Implementing Sim-nML if construct

description of a particular instruction. C++ facilitates efficient flattening of hierarchy either at compile time or at run time. The compile time flattening depends on C++ templates and run time flattening depends on inheritance polymorphism and C++ virtual functions.

■ Templatization

To describe how the class hierarchy is generated, we define some terms used, in the description below.

- *Instruction-tree* - A tree with *and-rules* as nodes in the tree. The relationships given by *or-rules* and the parameters of *and-rules* are represented as edges of the tree.

- *and-tree* - A skewed *instruction-tree* produced by *and-rules* only.
- *or-tree* - A non-skewed *instruction-tree* that has at least one *or-rule*. The branches due to *or-rule* will have different images.

The class hierarchy is generated with the help of following guidelines. The quality of code generated depends on users input.

- A class is generated for each *and-rule*.
- If instruction tree of a parameter of an *and-rule* is an *and-tree* or if the type of the parameter is pre-defined type then an object member of that type is created. See Figure 19.
- If instruction tree of a parameter of an *and-rule* is an *or-tree* then the class is templated for that parameter. See Figure 19. In Figure 19, to create an multiply instruction class we will instantiate by,

```
instruction < multiply >
```

Figure 19 shows part of the code generated for the simple Sim-nML description in Figure 3, Chapter 3. Multiple inheritance and parameterization helps to generate the code precisely and concisely. C++ function inlining feature makes this implementation as efficient as a non-hierarchical implementation.

■ *Flattening using Polymorphism*

This method is highly dependent on the inheritance polymorphism supported by C++. In this method, each *or-rule* is converted into an abstract class which declares the polymorphic functions. The class generated for *and-rule* are derived from the class generated for the parent *or-rule*. Figure 20 shows part of code generated for the simple Sim-nML description in Figure 3, Chapter 3. At the run time, depending on the opcode, we can decide whether the present instruction is a multiply instruction or plus instruction. If it is a multiply instruction then we can create the object by invoking `instruction` class constructor with a pointer to newly created multiply object.

```

class plus {
    ...
};
class multiply {
    ...
};
template < class T0 >
class instruction : public instr_base {
private:
    image_type  _image_;
    T0  x;
    byte      data;

public:
    nml_string syntax ( ) { ... }
    nml_string image ( ) { ... }
    void action ( )
    {
        tmp = data;
        x.action ( );
    }
    void preact ( )
    {
        PC = PC + 2;
    }
    void uses ( uses_sequence & _u_ ) { ... }
    instruction ( image_type  _img_ = 0, int _start_ = 0 )
                : x ( _image_, _start_ + 2 )
    {
        int _pos_ = 0;
        _pos_ += _start_ + 2 + x.image().size() + 1;
        data = extract_bit_fields ( _image_, _pos_, 8 );
        _pos_ += 8;
    }
};

```

Figure 19: Templatized Class Hierarchy Generated

```

class binaction {
public:
    virtual nml_string syntax ( ) = 0;
    virtual nml_string image ( ) = 0;
    virtual void action ( ) = 0;
    virtual int uses ( ) = 0;
};
class plus : public binaction {
    ...
};
class multiply : public binaction {
    ...
};
class instruction {
    binaction & x;
    ....
public:
    ...
    instruction ( binaction * b, image_type _img_ = 0,
                  int _start_ = 0 ) : x ( *b )
    {
        ...
    }
};

```

Figure 20: Class Hierarchy Using Polymorphism

The disadvantage of templatization method is that the generated object code size explodes as the classes are instantiated and the hierarchy is flattened. The disadvantage of polymorphism method is that invoking nested calls of virtual functions leads to high performance overhead. We hope the hybrid model will work the best. But, selecting an alternative approach in hybrid model should employ some heuristic.

4.2 Implementation of Simulator Generator

The objective of simulator generator is to generate C++ class hierarchies with **uses** function definitions from Sim-nML specification of a processor model. This constitutes

the first phase in simulator building effort. Once the class hierarchies are generated, these classes can be linked with other generic simulator libraries to produce a complete simulator. The simulator generator is implemented in three phases. Below, we describe the objective and implementation of each phase.

4.2.1 Phase 1 - Gathering Hierarchical Information

The objective of this phase is to construct the *op-rule* tree, find the type of the subtrees, structure the tree data so that it will be useful in code generation stage and to find the type of the simple expressions. The structured tree data is written to an intermediate file. This file is taken as the input along-with the Sim-nML description during the code generation stage.

Sim-nML attribute can be a function, for example, an *action* attribute, or can be an expression. The expression attributes can be implemented as functions returning the value of the expression. Since, C++ is a strongly typed language, the type of the return value should be declared, Therefore, type detection of attribute expressions are important. The first phase is useful for detecting the type of the simple expressions

```
op add_immediate_carry ( x : add_immediate )
value1 = carry_flag + x.value2
...
op add_immediate ( data : card ( 8 ) )
value2 = data
...
```

Figure 21: Type of Attributes

such as value2 in Figure 21. But, the type of value1 in Figure 21 can not be detected as this depends on the type of value2.

4.2.2 Phase 2 - Type Detection

The main objective of this phase is to detect the type of expressions for which it is unknown. The intermediate file produced by the Phase 1 is taken as an input to

this phase along-with the Sim-nML description and an intermediate file of the same format as in Phase 1, is generated as output. For example, in Figure 21, since the type of value2 is known already, the type of value1 can be detected in this phase. This phase is repeated till the type of all attributes are found.

4.2.3 Phase 3 - Code Generation

This phase takes the intermediate file generated by the Phase 2 and the Sim-nML description as input and generates the C++ class hierarchies as described in Section 4.1. The present implementation follows the design described in Design - 1 of Section 4.1.1 to generate a sequence of `uses_nodes` and uses templatization to flatten the instruction hierarchy. In addition, this phase generates a decode file which will be helpful for creating appropriate instruction objects. The decode file for the example in Figure 3, Chapter 3, is shown Figure 22. In the decode file, * indicates don't care,

11000000*****	1
11000001*****	2

Figure 22: Decode File

i.e., either of 0 or 1 can match for *. From the decode file, it is easy to generate the opcode and the opcode mask. The numbers opposite to each opcode field is useful to jump to proper location using a switch statement. The opcode file in conjunction with a decoding function which is generated along-with the class hierarchy, is useful to create appropriate instruction object and the sequence of `uses_nodes`. Figure 23, shows the code generated for decode function for the Sim-nML description in Figure 3, Chapter 3. In Figure 23, `pair< instr_base*, uses_sequence* >` indicates a structure containing two pointers and `image_choice` is nothing but the number got from decode file. The code generation phase generates the C++ template classes and writes it to an include file. This file is included wherever the class definitions are required. C++ compiler instantiates the required classes and functions during the compile time.


```

pair< instr_base*, uses_sequence* >
init_instr ( int image_choice, image_type image ){
    switch ( image_choice ) {
        case 1: {
            instruction< plus > *ptr;
            ptr = new instruction< branch_instr > ( image );
            uses_sequence * uses_ptr;
            uses_ptr = new uses_sequence;
            ptr -> uses ( *uses_ptr );
            pair< instr_base*, uses_sequence* > p ( ptr, uses_ptr );
            return p;
        }
        case 2: {
            instruction< multiply > *ptr;
            ptr = new instruction< multiply > ( image );
            uses_sequence * uses_ptr;
            uses_ptr = new uses_sequence;
            ptr -> uses ( *uses_ptr );
            pair< instr_base*, uses_sequence* > p ( ptr, uses_ptr );
            return p;
        }
    }
}

```

Figure 23: Decode Function

4.3 Implementation of nML Type System

With C++'s extensible type system, it is easy to implement nML types. The nML *card* type is implemented with the unsigned long data type of C++. The nML *int* type is implemented with the long data type. At present, the size of nML types are restricted by the size of long which is machine dependent. The type of result of an operation is decided by the rules described in nML[2] specification. The carry generated by an immediate operation is stored in a global variable named `_GLOBAL_CARRY_`. The present implementation emulates *float* with the help of C++'s `double` and `float`. Therefore, *float* of sizes other than C++'s `float` and `double` are not supported. nML's *fixed* point types are emulated with the same sized *card* types.

4.4 Generic Simulator Library

The present implementation of generic simulator library contains routines to implement a cycle based simulator. The library implements a resource manager which keeps track of the blocked resources. The library provides functions to block and free a sequence of resources. The library maintains a list of active instructions. In each clock, `fetch_instr` function fetches the instruction pointed by PC. Then the `fetch_instr` function decodes the image with the help of the decode file to find the value of `image_choice` variable in `init_instr` function as shown in Figure 23. After decoding, the appropriate instruction class and the corresponding `uses_sequence` is built with the help of `init_instr` function. If the first set of resources used by an instruction are available, then the newly fetched instruction is added to the active instruction list. Otherwise, the instruction is dropped. The instruction fetch operation is repeated till the `fetch_instr` fails to add a new instruction to the active instruction list due to non-availability of first set of resources. After fetch is stopped, each instruction in the active list is clocked. The clocking is done in FIFO order. This implements the Sim-nML semantics that the resources are allocated in FIFO order correctly.

Chapter 5

Results and Conclusion

In this chapter, we describe some experimental performance models specified in Sim-nML. We describe the way to generate performance simulator from Sim-nML specification and discuss the performance issues of generated simulator. We then give a brief conclusion to our work and give some guidelines for future work.

5.1 Performance Models with Sim-nML

As part of the thesis work, we developed performance models for two processors using Sim-nML. The first one is a performance model of a hypothetical superscalar processor employing branch prediction to reduce branch penalties. The Sim-nML specification for this model is described in Chapter 3. The second one is a performance model of a DEC Alpha 211064 processor. We have modeled only a partial set of instructions. This set is sufficient to encode applications such as bubble sort.

5.1.1 Generating Performance Simulator

The complete procedure to generate a performance simulator from Sim-nML specification is encapsulated into a shell script. When this script is run with Sim-nML specification file as argument, the various phases of simulator generator are executed sequentially. These phases create some intermediate files to pass the information to

the next phase. The final phase of the simulator generator generates the template classes, decode file, decode function as described in Chapter 4. The generated code is linked with the generic simulator library to obtain the performance simulator.

5.1.2 Results

■ *Hypothetical Superscalar Processor*

The hypothetical superscalar processor model was tested with a small loop which multiplies the nonzero initial value of Count register with 4 and adds 3 to it. The result is accumulated in AC. The assembly code for this small program is shown in Figure 24. When simulation is done, in each loop the conditional branch at line

1	add	3	
2	loop: add	4	
3	dec_count		
4	brnz	loop	
5	add	0	#NOP

Figure 24: Multiplication By Addition

number 4 is predicted. The model uses always taken branch prediction policy. When branch is mispredicted the branch error exception is invoked and the instruction that is fetched from the wrong path, i.e., add instruction at line 2, is disabled from writing the accumulator. Thus, the value is correctly computed.

The generated simulator can simulate about 500 instructions per second. At present, the generated performance simulator captures partial simulated instruction context using `uses_node` as described in Method - 1 in Chapter 4. Moreover, the `uses_node` list for an instruction is constructed whenever the instruction enters the processor pipeline. For example, if loop mentioned above is simulated 1000 times then `uses_node` list for `dec_count` is constructed for 1000 times. This leads to heavy performance penalty.

We experimented by changing the generated code by hand and removed the aforementioned penalty. The instrumented simulator simulates upto 3000 instructions per second. In search of improving the performance, we changed the generated code by hand in such a way that it captures the partially simulated instruction context in state variables as described in Method - 2 in Chapter 4. The simulator thus generated simulates upto 4000 instructions per second.

■ *Alpha Processor*

The Alpha processor model is tested with the help of a bubble sort program which sorts an array of 100 unsigned numbers. The program is written in the assembly. The instructions used in this program are modeled. The model includes the hierarchical memory system and TLB lookahead. The unresolved conditional branches are predicted with always taken policy. The generated simulator simulates around 400 instructions per second.

5.2 Conclusion

The ever-increasing complexity of modern processors and the growing trend towards specification driven system design automation has necessitated design automation tools with high level of abstraction. Performance modeling plays a major role in the specification driven system design. The level of abstraction provided by conventional languages such as C, VHDL etc. is not sufficient for fast prototyping of performance models and its simulation.

In this thesis work, we developed a language Sim-nML which is helpful for performance modeling. Sim-nML is essentially an extension of nML machine description formalism. It is simple, elegant and powerful enough to model machine behavior at instruction level. Sim-nML is capable of specifying the control flow in a clear way using resource usage model. Sim-nML language can be used for automatic generation of compilers, assemblers, disassemblers, instruction set simulators.

As part of the thesis work, we have designed and implemented an instruction set

simulator generator which is helpful as performance analysis automation tool. This simulator generator takes Sim-nML specification as input and generates C++ code for performance simulator. The generated code is powerful enough to emulate the inherent parallelism of complex modern processors.

We have demonstrated the power of the Sim-nML specification and the simulator generator with the help of experimental performance models. Although these models are quite small comparing to practical systems, these models capture most of the complex concepts such as branch prediction present in modern practical systems.

5.3 Future Work

The following future work can be undertaken as an extension to this project. All these extensions center around further improving the performance of the generated simulator.

- i. **Hybrid Simulator** - The state of art performance simulators can simulate around 10,000 to 20,000 instructions per second. At present, generated simulator is entirely cycle-based. Further, a simulator that implements only event-based simulation will incur high overhead due to event-management because instructions frequently lock the resources for one unit of time. We can improve the performance if we resolve to hybrid approach in which a waiting list of instructions is maintained for some resources which are locked frequently for more than one clock cycles. These resources can be found statically from Sim-nML specification.
- ii. **Efficient Flattening of Hierarchy** - Both approaches described for flattening the hierarchy in Chapter 4, viz. Templatzation, Polymorphism, have advantages as well as disadvantages. Therefore, if a hybrid approach is used which mainly captures the advantages of both of these approaches then this will be helpful in boosting the performance of the simulator.
- iii. **Multiple Performance Models** - At present, the generated simulator simulates a single performance model. In practice, it is desirable to have more than one

model. For example, two performance models, first model which is fast and does not include features such as pipelining, is useful at the beginning of the simulation to warm up the caches. While the second model is slow and simulates all architectural details to perfection, is useful to analyze the design from an interesting point.

Bibliography

- [1] DIEP, T. A., AND SHEN, J. P. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer* (Dec 1995), 57–64.
- [2] FREERICK, M. The nML Machine Description Formalism, 7 1993.
http://www.cs.tu-berlin.de/~mfx/dvi_docs/nml_2.dvi.gz.
- [3] MENDEL ROSENBLUM, E. BUGNION, S. D., AND HERROD, S. A. Using the Simos Machine Simulator to Study Complex Computer Systems. *ACM Trans on Modeling and Computer Simulation* 7, 1 (Jan 1997), 78–103.
- [4] PAITHANKAR, A. AINT: A Tool for Simulation of Shared Memory Multiprocessors, 1996. Master's Thesis, Univ. of Colorado, Boulder, Colo.
- [5] POURSEPANJ, A. The PowerPC Performance Modeling Methodology. *Comm. ACM* (Jan 1994), 47–55.
- [6] REILLY, M., AND EDMONDSON, J. Performance Simulation of an Alpha Microprocessor. *IEEE Computer* 31, 5 (May 1998), 50–58.
- [7] SRIVASTAVA, A., AND EUSTACE, A. ATOM: A System for Building Customized Program Analysis Tools. In *ACM SIGPLAN Conf. Programming Language Design and Implementation* (New York, 1994), ACM Press.
- [8] STROUSTRUP, B. *The C++ Programming Language*. Addison-Wesley, Massachusetts, 1995.
- [9] SUBHASIS LAHA, J. P., AND IYER, R. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans on Computers* 37, 11 (Nov 1988), 1325–1336.

- [10] WANG, W.-H., AND BAER, J.-L. Efficient Trace-Driven Simulation Methods for Cache Performance Analysis. *ACM Trans on Computer Systems* 9, 3 (Aug 1991), 222–241.